How Perspective-Based Reading Can Improve Requirements Inspections

Perspective-based reading gives developers a set of procedures to inspect software products for defects. Detecting and correcting these defects early in the development process can save a lot of time and money, and possibly avoid some embarrassment.

Forrest Shull
Ioana Rus
Victor Basili
Fraunhofer Center for Experimental Software Engineering, Maryland

Because defects constitute an unavoidable aspect of software development, discovering and removing them early is crucial. Overlooked defects (like faults in the software system requirements, design, or code) propagate to subsequent development phases, and detecting and correcting them becomes more difficult.[1] At best, developers will eventually catch the defects, but at the expense of schedule delays and additional product-development costs. At worst, the defects will remain, and customers will receive a faulty product.

Providing complete, consistent, unambiguous, and correct documents throughout the lifecycle improves the chances for a higher-quality software system. Therefore, checking software documents for defects before proceeding to the next development phase contributes to overall system quality. An inspection by qualified personnel is the best way to accomplish this. Robert Glass stated that "inspections, by all accounts, do a better job of error-removal than any competing technology (that is, inspections tend to find more errors), and they do it at a lower cost (the cost per error found is lower)".[2] Most published work shows that inspections are both effective and efficient.[3] In particular, inspections in the requirements specification phase[4,5] can catch inconsistent or incorrect requirements for the system before they form the basis for design or implementation, which would necessitate rework.

Perspective-based reading provides a set of procedures that can help developers solve software requirements inspection problems. PBR reviewers stand in for specific stakeholders in the document (such as designers or testers) to verify the quality of requirements specifications. A PBR review ensures that requirements are sufficient to support all the necessary later stages of software development. PBR offers several benefits compared with other inspection approaches, and development organizations can customize PBR to fit their needs.

TECHNIQUES FOR IMPROVING INSPECTIONS

Proper inspection implementation requires an accurate understanding of the related tasks and organization contexts and of the roles of those conducting the inspection.[5] Usually, the inspection process has several phases: planning, overview, defect detection, defect collection, defect correction, and follow-up.

The defect detection phase is the most significant. Reviewers (typically software developers) read a software document and apply some technique (formal or informal) to uncover defects. The "Techniques for Reading Requirements Documents" sidebar provides a brief comparative description of techniques reviewers can use to read requirements documents.

A complete description of inspections must address five dimensions:[3]

- technical
- managerial
- organizational
- assessment
- tool support

0018-9162/00/$10.00 © 2000 IEEE

Techniques for Reading Requirements Documents

Several techniques exist for individually reviewing requirements documents. At one extreme is the *ad hoc* review, a review with no formal, systematic procedure, based only on individual experience. A *checklist* review makes the inspection process slightly better-defined by providing reviewers with a list of items on which to focus. *Defect-Based Reading (DBR)* provides a set of systematic procedures that reviewers can follow, which are tailored to the formal Software Cost Reduction (SCR) notation. Like DBR, *Perspective-Based Reading (PBR)* is a scenario-based technique that provides procedural guidance, tailored to requirements expressed in natural language (for example, English).

Table A presents some characteristics of these techniques. We present the requirements languages for which each technique is usable and compare them according to the following criteria:

- *Systematic.* Are the specific steps of the individual review process definable?
- *Focused.* Must different reviewers focus on different aspects of the document?
- *Allows controlled improvement.* Based on feedback, can reviewers identify and improve specific aspects of the technique?
- *Customizable.* Can reviewers customize the technique to a specific project or organization?
- *Allows training.* Can reviewers use a set of steps to train themselves in applying the technique?

Table A. Characteristics of requirements reading technique.

Technique	Requirements language	Systematic?	Focused?	Controlled improvement?	Customizable?	Training?
Ad hoc	Any	No	No	No	No	No
Checklist	Any	Partially	No	Partially	Yes	Partially
Defect- based reading	Software cost reduction (formal notation)	Yes	Yes	Yes	Yes	Yes
Perspective-based reading	Natural language	Yes	Yes	Yes	Yes	Yes

Choosing to focus on organizational aspects such as the number of participants and the structure and frequency of meetings, many publications[6,7] have neglected the technical dimension—the review techniques and how reviewers actually use them.

However, technical considerations are important, and significant problems in this area need attention. While reviewers may know how to *write* software documents, they may have little expertise in *reading* them. Reviewers typically rely on ad hoc reading techniques, with no well-defined procedure, learning largely by doing and gaining significant expertise only gradually. The difficulty in providing training for a poorly defined or undefined process such as an ad hoc review further compounds the problem. For large software projects, improving the review process requires understanding what defects escaped the review and targeting them more effectively. Letting every reviewer develop a review process makes the communication of effective review strategies more difficult, hampering the widespread dissemination of developed expertise.

A software reading technique provides a concrete set of instructions that explain how to read a software document and what a reviewer should look for.[5] Reviewers can use these guidelines during the inspection's preparation phase to examine a given software document for defects. Rather than leaving reviewers to their own devices, as in ad hoc reviews, software reading techniques collect knowledge about best practices for defect detection into a single procedure.

Researchers at the Experimental Software Engineering Group at the University of Maryland, College Park created PBR to provide a set of software reading techniques for finding defects in English-language requirements documents. Widely applicable and customizable to particular situations, PBR is not strictly formalized into a definitive set of procedures. Rather, using PBR entails

- selecting a set of perspectives for reviewing the requirements document;
- creating or tailoring procedures for each perspective usable for building a model of the relevant requirements information;
- augmenting each procedure with questions for finding defects while creating the model; and
- applying the procedures to review the document.

PBR OVERVIEW

PBR helps reviewers answer two important questions about the requirements they inspect: What information in these requirements should they check? How do they identify defects in that information?

Most inspection techniques do not help the reader focus on a particular aspect of the document. They treat all requirements information as equally important because it all represents real functional requirements and constraints on that functionality. But then document reviewers end up with ill-defined responsibilities for finding all types of defects in the entire document.

Different perspectives

PBR operates under the premise that different information in the requirements is more or less important for the different uses of the document. Many different people use a requirements document to support tasks throughout the development life cycle. Conceivably, each person finds different aspects of the requirements important for accomplishing a particular task. Therefore, PBR provides a set of individual reviews, each from a particular requirements user's point of view, that collectively cover the document's relevant aspects.

PBR requires identifying the users of a specific software artifact (here, the requirements). This process is similar to constructing system use cases, which requires identifying who will use the system and in what way. This selection of users varies according to organization or project needs. In our environment, we identified three major uses of the requirements at later stages of the life cycle:

- *A description of the customer's needs*. The requirements describe the set of functionality and performance constraints that the final system must meet.
- *A basis for the system design*. The system designer must create a design that can function according to the requirements and within the allowed constraints.
- *A point of comparison for system test*. The system's test plan must ensure that the software correctly implements functionality and performance requirements.

These uses suggest perspectives for reviewing the requirements document: The designer needs correct requirements with sufficient detail for the major system components under review. The tester, concerned about requirements testability, needs to see sufficient detail to construct test plans. The customer (or user) of the system requires that the requirements completely and correctly capture the necessary system functionality. A failure to satisfy any of these needs constitutes a *requirements defect*—a deficiency in the requirements quality that can hamper software development.

Other perspectives can lead to defect detection as well; the examples we've given simply represent the perspectives we've experienced. Depending on the environment in which users apply PBR, they may find a different set of perspectives more applicable. For example, a system expected to have a long operational life span could also be reviewed from a maintainer's perspective (which would be concerned with verifying that requirements are easily extensible). Deciding on the most appropriate set of perspectives is one way of tailoring PBR to a particular environment.

Thus, in a PBR inspection, each reviewer on a team assumes a specific user's perspective. The reviewer creates a high-level version of the work products typical of what the user would normally produce. From the designer, tester, and customer perspectives, the relevant work products would be a high-level system design, a system test plan, and an enumeration of the described functionality, respectively.

The objective is to avoid duplicating work during the software development process by creating representations of the system capable of supporting two distinct types of activities. First, reviewers can use the representations as a basis for the later creation of more specific work products. Second, reviewers use the representations to analyze how well the requirements can support the necessary tasks.

Because PBR works from a definite procedure, and not the reviewer's own experience with recognizing defects, new reviewers can receive training in the procedure's steps.

An appropriate level of detail helps reviewers construct the relevant system representations. Organizations vary these levels to further tailor PBR. When used by more experienced reviewers, the PBR procedures should rely mainly on a reviewer's previous experience in creating design plans, test plans, and user manuals. Alternately, a reviewer can select a very specific type of representation for each perspective (for example, structured design, equivalence partitioning test plans, and use cases, respectively), more specific procedures are appropriate for less experienced reviewers.

Using different perspectives offers a number of beneficial attributes:

- *Systematic*. Explicitly identifying the different uses for the requirements gives reviewers a definite procedure for verifying whether those uses are achievable.
- *Focused*. PBR helps reviewers concentrate more effectively on certain types of defects, rather than having to look for all possible types. A study of PBR's effectiveness[5] showed that this additional focus helped reviewers find more defects than if they used a less structured approach, even though they were concentrating on specific aspects of the document while using PBR. One reviewer summarized this benefit by saying, "It really helps to have a perspective because it focuses my questions. I get confused trying to wear all the hats!" Additionally, this focus helps avoid duplicated effort among team members.
- *Goal-oriented and customizable*. Reviewers select the perspectives used by PBR to reflect the requirements' uses. In a new organization, the perspectives change to reflect how that organization uses its requirements documents and the inspection's specific goals. Because each perspective gives a specific procedure, reviewers can tailor the procedure to the organization's needs. For

example, the procedure can be more or less specific based on the reviewers' expertise.
- *Transferable via training*. Because PBR works from a definite procedure, and not the reviewer's own experience with recognizing defects, new reviewers can receive training in the procedure's steps. Additionally, because PBR uses work products for other life-cycle stages, reviewers can apply their training and experience to tasks that may seem more natural to them.

Identifying defects

Once reviewers have created relevant representations of the requirements, they still need to determine what defects may exist. PBR techniques provide questions tailored to each step of the procedure for creating the representation. As the reviewers construct the representation, they answer a series of questions about the work products. Requirements that do not provide enough information to answer the questions usually do not provide enough information to support the user. Thus, reviewers can identify and fix defects beforehand, so that requirements are ready to support that task later in the product life cycle.

We defined a taxonomy of requirements defects to assure the sufficiency of the questions at each phase, and we developed the PBR questions to detect each type of defect. We based this taxonomy on the IEEE definition of necessary attributes of good requirements documents,[8] and our definition is similar to others that help track requirements defects.[9] Table 1 lists the types of defects that PBR helps detect. In practice, these classifications are not orthogonal; a given defect could conceivably fit into more than one category, depending on the interpretation of the classifier. Nor is this taxonomy definitive; an organization can add other categories, depending on its needs.

The sidebar, "A Sample PBR Procedure," presents an example of how a tester uses the PBR procedure. (We provide additional instantiations for other perspectives at http://fc-md.umd.edu/reading/reading.html.) This example includes a series of questions tailored to each step of the procedure for developing an equivalence-partitioning test plan. For example, Part b includes guidelines for identifying test cases for each equivalence set. A series of questions check for missing information (Qb.1), ambiguous information (Qb.2), inconsistencies between requirements (Qb.3), and incorrect facts (Qb.4). The question list does not include defects of extraneous information or other miscellaneous reasons (such as document structure) because we did not think them relevant for this step.

Overall, PBR's detailed questions have the following benefits:

- *Allow controlled improvement*: Reviewers can reword, add, or delete specific questions.
- *Allow training*: Reviewers can train to better understand the parts of a representation or work product that correspond to particular questions.

APPLYING PBR

PBR does not predicate a specific format for the requirements. We tailored it to requirements that use

A Sample PBR Procedure

For each requirement, generate a test or set of test cases that let you ensure that a system implementation satisfies the requirement. Follow the procedure below to generate the test cases, using the questions provided to identify faults in the requirements.

General questions

Read each requirement once and record the number and page on the form provided, along with the inputs to the requirement.

- Q1. Does the requirement make sense from what you know about the application or from what is specified in the general description?
- Q2. Do you have all the information necessary to identify the inputs to the requirement? Based on the general requirements and your domain knowledge, are these inputs correct for this requirement?
- Q3. Have any of the necessary inputs been omitted?
- Q4. Are any inputs specified that are not needed for this requirement?
- Q5. Is this requirement in the appropriate section of the document?

Part a: Building equivalence sets.

For each input, divide the input domain into sets of data (called equivalence sets); all values in each set will cause the system to behave similarly. Determine the equivalence sets for a particular input by understanding the sets of conditions that affect the requirement's behavior. You may find it helpful to keep the following guidelines in mind when creating equivalence classes:

- If an input condition specifies a range, at least one valid (the set of values in the range) and two invalid equivalence sets (the set of values less than the lowest extreme of the range, and the set of values greater than the largest extreme) are defined.
- If an input condition specifies a set's member, at least one valid (the set itself) and one invalid equivalence set (the valid set's complement) is defined.

Table 1. Requirements defects that PBR helps detect.

Missing information	Any significant requirement related to functionality, performance, design constraints, attributes, or external interface not included
	Undefined software responses to all realizable classes of input data in all realizable classes of situations
	Sections of the requirements document
	Figure labels and references, tables, and diagrams
	Definitions of terms and units of measure
Ambiguous information	Multiple interpretations caused by using multiple terms for the same characteristic or multiple meanings of a term in a particular context
Inconsistent information	Two or more requirements that conflict with one another
Incorrect fact	A requirement-asserted fact that cannot be true under the conditions specified for the system.
Extraneous information	Unnecessary or unused information (at best, it is irrelevant; at worst, it may confuse requirements users)
Miscellaneous defects	Other errors, such as including a requirement in the wrong section

English (or some other natural language) to describe the system functionality, but users could easily tailor it to suit a formal requirements specification language, such as Software Cost Reduction.

Building on experience

PBR does not assume that developers already possess the skills for effectively analyzing requirements documents. Rather, PBR helps developers build on their experience in other phases of software development, letting them use skills they already have (such as for creating designs, test plans, or user manuals) to help them better understand requirements documents. Thus, some reviewers (especially less experienced ones) may find that PBR appears to be a more natural approach to reviewing requirements than other methods, such as checklists. For this reason, PBR's usefulness may lie in how it trains inexperienced reviewers. Of course, reviewers must at least have *some* experience or training with the techniques for creating the relevant work products they'll have to apply (such as creating test plans).

An organizational culture accustomed to performing reviews is useful for effective application of PBR. Novice reviewers who have no experience with looking for defects are not used to thinking about the effect of faulty requirements on later stages of the life cycle. Therefore, conveying to novice reviewers the importance of providing a definition of a requirements defect sufficiently detailed for identification is difficult. PBR attempts to provide guidelines in this area, and it uses the defect taxonomy to flag problem areas for reviewers. But reviewers still need to build up their experience in this area.

- If an input condition requires a specific value, then one valid (the set containing the value itself) and two invalid equivalence sets (the set of values less than and the set greater than the value) are defined.

Each equivalence set should be recorded in the spaces provided on the form under the appropriate input.

- Qa.1. Do you have enough information to construct the equivalence sets for each input? Can you specify the boundaries of the sets at an appropriate level of detail?
- Qa.2. According to the information in the requirements, are the sets constructed so that no value appears in more than one set?
- Qa.3. Do the requirements state that a particular value should appear in more than one equivalence set (that is, do they specify more than one type of response for the same value)? Do the requirements specify that a value should appear in the wrong equivalence set?

Part b: Testing equivalence sets.

For each equivalence set, write test cases, and record them beneath the associated equivalence set on the form. Select typical test cases as well as values at and near the boundaries of the sets. For example, if the requirement expects input values in the 0 to 100 range, the test cases selected might be 0, 1, 56, 99, and 100. Finally, for each equivalence set, record the expected resulting behavior (that is, how do you expect the system to respond to the test cases you just made up?)

- Qb.1. Do you have enough information to create test cases for each equivalence set?
- Qb.2. Are there other interpretations of this requirement that the implementer might make on the basis of the description given? Will this affect the tests you generate?
- Qb.3. Is there another requirement for which you would generate a similar test case but would get a contradictory result?
- Qb.4. Can you be sure that the tests generated will yield the correct values in the correct units? Is the resulting behavior specified appropriately?

PBR leads to improved defect detection rates for both individual reviewers and review teams working with unfamiliar application domains.

Having reviews in place also means that an organization has probably already dealt with the difficulties inherent in reviews themselves, such as motivating reviewers to perform satisfactory reviews and scheduling team meetings as part of the development cycle.

Perhaps the most serious constraint on the use of PBR is the amount of reviewer effort required. The improved rate of defect detection comes at the price of a higher amount of effort on the part of the reviewers. For example, in a study of software professionals,[5] the average review time required for PBR ranged from almost the same as for the professionals' usual approach to 30 percent more, depending on the document being reviewed. However, much of this extra effort actually produces the high-level representations of the system that may save effort at later stages of the life cycle. For example, the high-level test plan developed during PBR can serve as the basis for the actual test plan used to evaluate the implementation.

Lessons learned

Researchers in the U.S.,[10] Norway,[11] and Germany[12] have conducted studies using more than 150 software engineering students in university classes to evaluate and evolve PBR techniques. These students range from undergraduates with little previous review experience to professionals with more than 20 years experience in industry who were returning for advanced degrees. In 1995 and 1996, researchers from the Experimental Software Engineering Group ran studies using 25 professional developers from the NASA Goddard Space Flight Center.[5] These developers first applied the requirements review technique they used at NASA, and then were trained in PBR and applied the new technique to a similar requirements document. In this way, the researchers could assess how well these professionals performed using PBR compared with how they performed using their usual review technique.

These studies supported the notion that PBR leads to improved defect detection rates for both individual reviewers and review teams working with unfamiliar application domains. These studies also showed that, for a familiar application domain, experienced reviewers sometimes ignore the PBR procedure and revert to using previously acquired heuristics. Training and reinforcement can overcome this tendency.

By observing the use of PBR in varied environments by several reviewers, these studies also helped researchers better understand the effects of PBR in different contexts and for different types of users. For example, PBR seems best suited for reviewers with a certain range of experience. Reviewers who have previously inspected requirements documents on multiple industrial projects have, over time, typically developed their own approaches and do not benefit significantly from the introduction of PBR. Reviewers who have little experience (those who have never trained, or have trained but never applied their skills on a real project) with the relevant representations (such as designs or test plans) need to receive sufficient training before they can effectively apply PBR.[10] This training seems necessary so that the difficulties of creating the representation of the system do not distract from the process of checking for defects.

PBR provides a framework that represents an improved approach for conducting requirements reviews. However, such an approach will only be effective when an organization tailors the framework to its own needs and uses feedback from its reviewers to continually improve and refine the techniques.

Studies have shown that development teams that use PBR to inspect requirements documents tend to detect more defects than they do using other, less structured approaches. Although PBR requires more effort, it offers a number of advantages compared with current practices. Relatively novice reviewers can use PBR techniques to apply their expertise in other development tasks to defect detection. Using PBR improves team meetings by helping team members build up expertise in different aspects of a requirements document. It creates high-level representations of the software system, usable as a basis for work products in later stages of development. Each development organization can customize PBR's set of perspectives, level of detail, and types of questions. PBR facilitates controlled improvement, providing a definite procedure, alterable according to project metrics and reviewer feedback. ✹

References

1. J.C. Kelly, J.S. Sherif, and J. Hops, "An Analysis of Defect Densities Found During Software Inspections," *J. Systems Software*, Feb. 1992, pp. 111-117.
2. R.L. Glass, "Inspections—Some Surprising Findings," *Comm. ACM*, Apr. 1999, pp. 17-19.
3. O. Laitenberger and J. DeBaud, *An Encompassing Life-Cycle Centric Survey of Software Inspection,* Tech. Report No. ISERN-98-32, Fraunhofer Inst. Experimental Software Eng., Kaiserslautern, Germany, 1998; http://www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-98-32.pdf.
4. A. Porter, L. Votta Jr., and V. Basili, "Comparing Detection Methods for Software Requirements Inspections: A Replicated Experiment," *IEEE Trans. Software Eng.*, June 1995, pp. 563-575.
5. V. Basili et al., "The Empirical Investigation of Perspec-

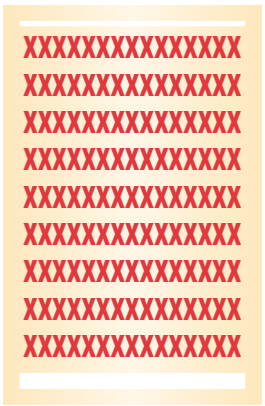
tive-Based Reading," *Empirical Software Eng.: An Int'l J.*, Vol. 1, No. 2, 1996, pp. 133-164
6. M. Fagan, "Advances in Software Inspections," *IEEE Trans. Software Eng.*, July 1986, pp. 744-751.
7. T. Gilb and D. Graham, *Software Inspection*, Addison-Wesley Longman, Reading, Mass., 1993.
8. *IEEE Guide to Software Requirements Specifications*, ANSI/IEEE Standard 830-1984, IEEE Press, Piscataway, N.J., 1984.
9. V. Basili and D. Weiss, "Evaluation of the A-7 Requirements Document by Analysis of Change Data," *Proc. 5th Int'l Conf. Software Eng., IEEE,* CS Press, Los Alamitos, Calif., Mar. 1981, pp. 314-323.
10. F. Shull, *Developing Techniques for Using Software Documents: A Series of Empirical Studies*, doctoral dissertation, Computer Science Dept., Univ. of Maryland, 1998; http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/postscript/shull_diss.ps.gz.
11. S. Sørumgård, *Verification of Process Conformance in Empirical Studies of Software Development*, doctoral dissertation, Dept. Computer and Information Science, Norwegian Univ. of Science and Technology, Trondheim, Norway, 1997; http://www.idt.unit.no/~sivert/ps/Thesis.ps.
12. M. Ciolkowski et al., *Empirical Investigation of Perspective-based Reading: A Replicated Experiment*, Tech. Report No. ISERN-97-13, Int'l Software Eng. Research Network, Kaiserslautern, Germany, 1997; http://www.iese.fhg.de/network/ISERN/pub/technical_reports/isern-97-13.pdf.

Forrest Shull *is a scientist for the Fraunhofer Center for Experimental Software Engineering, Maryland. His research interests include software process improvement, empirical software engineering, and software inspections. He received a PhD in computer science from the University of Maryland, College Park. Contact him at fshull@fc-md.umd.edu.*

Ioana Rus *is a scientist for the Fraunhofer Center for Experimental Software Engineering, Maryland. Her research interests include software process improvement, modeling and simulation, measurement and experimentation in software engineering, and artificial intelligence. She received a PhD in computer science and engineering from Arizona State University. Contact her at irus@fc-md.umd.edu.*

Victor Basili *is Executive Director of the Fraunhofer Center for Experimental Software Engineering and a professor in the Institute for Advanced Computer Studies and the Computer Sciences Department at the University of Maryland. His research interests include experimental software engineering and software quality assurance. He is a member of the IEEE and ACM Fellow. Contact him at basili@fc-md.umd.edu.*

XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX


XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX
XXXXXXXXXXXXXXX